progiter Documentation

Jon Crall

Jan 12, 2020

API

ProgIter lets you measure and print the progress of an iterative process. This can be done either via an iterable interface or using the manual api. Using the iterable inferface is most common.

ProgIter was originally developed independantly of `tqdm`, but the newer versions of this library have been designed to be compatible with tqdm-API. `ProgIter` is now a (mostly) drop-in alternative to **tqdm_**. The `tqdm` library may be more appropriate in some cases. *The main advantage of ''ProgIter'' is that it does not use any python threading*, and therefore can be safer with code that makes heavy use of multiprocessing. **'The reason'_** for this is that threading before forking may cause locks to be duplicated across processes, which may lead to deadlocks.

ProgIter is simpler than tqdm, which may be desirable for some applications. However, this also means ProgIter is not as extensible as tqdm. If you want a pretty bar, use tqdm; if you want useful information (rate, fraction-complete, estimated time remaining, time taken so far, current wall time) about your iteration by default, use progiter.

Example

The basic usage of ProgIter is simple and intuitive. Just wrap a python iterable. The following example wraps a `range` iterable and prints reported progress to stdout as the iterable is consumed. The `ProgIter` object accepts various keyword arguments to modify the details of how progress is measured and reported. See API documentation of the `ProgIter` classs here: https://progiter.readthedocs.io/en/latest/progiter.progiter.html#progiter.progiter.ProgIter

Note that by default ProgIter reports information about iteration-rate, fraction-complete, estimated time remaining, time taken so far, and the current wall time.

```
>>> from progiter import ProgIter
>>> def is_prime(n):
...     return n >= 2 and not any(n % i == 0 for i in range(2, n))
>>> for n in ProgIter(range(1000), verbose=2):
>>>     # do some work
>>>     is_prime(n)
    0/1000... rate=0.00 Hz, eta=?, total=0:00:00, wall=14:05 EST
    1/1000... rate=82241.25 Hz, eta=0:00:00, total=0:00:00, wall=14:05 EST
  257/1000... rate=177204.69 Hz, eta=0:00:00, total=0:00:00, wall=14:05 EST
  642/1000... rate=94099.22 Hz, eta=0:00:00, total=0:00:00, wall=14:05 EST
 1000/1000... rate=71886.74 Hz, eta=0:00:00, total=0:00:00, wall=14:05 EST
```

A Progress Iterator

ProgIter lets you measure and print the progress of an iterative process. This can be done either via an iterable interface or using the manual API. Using the iterable inferface is most common.

ProgIter was originally developed independantly of `tqdm`, but the newer versions of this library have been designed to be compatible with tqdm-API. `ProgIter` is now a (mostly) drop-in alternative to **tqdm_**. The `tqdm` library may be more appropriate in some cases. *The main advantage of ''ProgIter'' is that it does not use any python threading*, and therefore can be safer with code that makes heavy use of multiprocessing. **'The reason'_** for this is that threading before forking may cause locks to be duplicated across processes, which may lead to deadlocks.

ProgIter is simpler than tqdm, which may be desirable for some applications. However, this also means ProgIter is not as extensible as tqdm. If you want a pretty bar or need something fancy, use tqdm; if you want useful information about your iteration by default, use progiter.

The basic usage of ProgIter is simple and intuitive. Just wrap a python iterable. The following example wraps a `range` iterable and prints reported progress to stdout as the iterable is consumed.

Example

```
>>> for n in ProgIter(range(1000)):
>>>     # do some work
>>>     pass
```

Note that by default ProgIter reports information about iteration-rate, fraction-complete, estimated time remaining, time taken so far, and the current wall time.

Example

```
>>> # xdoctest: +IGNORE_WANT
>>> def is_prime(n):
...     return n >= 2 and not any(n % i == 0 for i in range(2, n))
>>> for n in ProgIter(range(1000), verbose=1):
>>>     # do some work
>>>     is_prime(n)
1000/1000... rate=21153.08 Hz, eta=0:00:00, total=0:00:00, wall=13:00 EST
```

For more complex applications is may sometimes be desireable to manually use the ProgIter API. This is done as follows:

Example

```
>>> # xdoctest: +IGNORE_WANT
>>> n = 3
>>> prog = ProgIter(desc='manual', total=n, verbose=3)
>>> prog.begin() # Manually begin progress iteration
>>> for _ in range(n):
...     prog.step(inc=1)  # specify the number of steps to increment
>>> prog.end()  # Manually end progress iteration
manual 0/3... rate=0 Hz, eta=?, total=0:00:00, wall=12:46 EST
manual 1/3... rate=12036.01 Hz, eta=0:00:00, total=0:00:00, wall=12:46 EST
manual 2/3... rate=16510.10 Hz, eta=0:00:00, total=0:00:00, wall=12:46 EST
manual 3/3... rate=20067.43 Hz, eta=0:00:00, total=0:00:00, wall=12:46 EST
```

When working with ProgIter in either iterable or manual mode you can use the `prog.ensure_newline` method to guarantee that the next call you make to stdout will start on a new line. You can also use the `prog.set_extra` method to update a dynamci "extra" message that is shown in the formatted output. The following example demonstrates this.

Example

```
>>> # xdoctest: +IGNORE_WANT
>>> def is_prime(n):
...     return n >= 2 and not any(n % i == 0 for i in range(2, n))
>>> _iter = range(1000)
>>> prog = ProgIter(_iter, desc='check primes', verbose=2)
>>> for n in prog:
>>>     if n == 97:
>>>         print('!!! Special print at n=97 !!!')
>>>     if is_prime(n):
>>>         prog.set_extra('Biggest prime so far: {}'.format(n))
>>>         prog.ensure_newline()
```

(continues on next page)

(continued from previous page)

```
check primes    0/1000... rate=0 Hz, eta=?, total=0:00:00, wall=12:55 EST
check primes    1/1000... rate=98376.78 Hz, eta=0:00:00, total=0:00:00, wall=12:55 EST
!!! Special print at n=97 !!!
check primes  257/1000...Biggest prime so far: 251 rate=308037.13 Hz, eta=0:00:00,␣
↪total=0:00:00, wall=12:55 EST
check primes  642/1000...Biggest prime so far: 641 rate=185166.01 Hz, eta=0:00:00,␣
↪total=0:00:00, wall=12:55 EST
check primes 1000/1000...Biggest prime so far: 997 rate=120063.72 Hz, eta=0:00:00,␣
↪total=0:00:00, wall=12:55 EST
```

CHAPTER 1

progiter

1.1 progiter package

1.1.1 Submodules

1.1.1.1 progiter.progiter module

A Progress Iterator

ProgIter lets you measure and print the progress of an iterative process. This can be done either via an iterable interface or using the manual API. Using the iterable inferface is most common.

ProgIter was originally developed independantly of `tqdm`, but the newer versions of this library have been designed to be compatible with tqdm-API. `ProgIter` is now a (mostly) drop-in alternative to **tqdm_**. The `tqdm` library may be more appropriate in some cases. *The main advantage of ‘‘ProgIter‘‘ is that it does not use any python threading*, and therefore can be safer with code that makes heavy use of multiprocessing. **‘The reason‘_** for this is that threading before forking may cause locks to be duplicated across processes, which may lead to deadlocks.

ProgIter is simpler than tqdm, which may be desirable for some applications. However, this also means ProgIter is not as extensible as tqdm. If you want a pretty bar or need something fancy, use tqdm; if you want useful information about your iteration by default, use progiter.

The basic usage of ProgIter is simple and intuitive. Just wrap a python iterable. The following example wraps a `range` iterable and prints reported progress to stdout as the iterable is consumed.

Example

```
>>> for n in ProgIter(range(1000)):
>>>     # do some work
>>>     pass
```

Note that by default ProgIter reports information about iteration-rate, fraction-complete, estimated time remaining, time taken so far, and the current wall time.

Example

```
>>> # xdoctest: +IGNORE_WANT
>>> def is_prime(n):
...     return n >= 2 and not any(n % i == 0 for i in range(2, n))
>>> for n in ProgIter(range(1000), verbose=1):
>>>     # do some work
>>>     is_prime(n)
1000/1000... rate=21153.08 Hz, eta=0:00:00, total=0:00:00, wall=13:00 EST
```

For more complex applications is may sometimes be desireable to manually use the ProgIter API. This is done as follows:

Example

```
>>> # xdoctest: +IGNORE_WANT
>>> n = 3
>>> prog = ProgIter(desc='manual', total=n, verbose=3)
>>> prog.begin() # Manually begin progress iteration
>>> for _ in range(n):
...     prog.step(inc=1)  # specify the number of steps to increment
>>> prog.end()  # Manually end progress iteration
manual 0/3... rate=0 Hz, eta=?, total=0:00:00, wall=12:46 EST
manual 1/3... rate=12036.01 Hz, eta=0:00:00, total=0:00:00, wall=12:46 EST
manual 2/3... rate=16510.10 Hz, eta=0:00:00, total=0:00:00, wall=12:46 EST
manual 3/3... rate=20067.43 Hz, eta=0:00:00, total=0:00:00, wall=12:46 EST
```

When working with ProgIter in either iterable or manual mode you can use the `prog.ensure_newline` method to guarantee that the next call you make to stdout will start on a new line. You can also use the `prog.set_extra` method to update a dynamci "extra" message that is shown in the formatted output. The following example demonstrates this.

Example

```
>>> # xdoctest: +IGNORE_WANT
>>> def is_prime(n):
...     return n >= 2 and not any(n % i == 0 for i in range(2, n))
>>> _iter = range(1000)
>>> prog = ProgIter(_iter, desc='check primes', verbose=2)
>>> for n in prog:
>>>     if n == 97:
>>>         print('!!! Special print at n=97 !!!')
>>>     if is_prime(n):
>>>         prog.set_extra('Biggest prime so far: {}'.format(n))
>>>         prog.ensure_newline()
check primes    0/1000... rate=0 Hz, eta=?, total=0:00:00, wall=12:55 EST
check primes    1/1000... rate=98376.78 Hz, eta=0:00:00, total=0:00:00, wall=12:55 EST
!!! Special print at n=97 !!!
check primes  257/1000...Biggest prime so far: 251 rate=308037.13 Hz, eta=0:00:00,␣
↪total=0:00:00, wall=12:55 EST
check primes  642/1000...Biggest prime so far: 641 rate=185166.01 Hz, eta=0:00:00,␣
↪total=0:00:00, wall=12:55 EST
check primes 1000/1000...Biggest prime so far: 997 rate=120063.72 Hz, eta=0:00:00,␣
↪total=0:00:00, wall=12:55 EST
```

class progiter.progiter.**ProgIter**(*iterable=None*, *desc=None*, *total=None*, *freq=1*, *initial=0*, *eta_window=64*, *clearline=True*, *adjust=True*, *time_thresh=2.0*, *show_times=True*, *enabled=True*, *verbose=None*, *stream=None*, *chunksize=None*, ***kwargs*)

Bases: progiter.progiter._TQDMCompat, progiter.progiter._BackwardsCompat

Prints progress as an iterator progresses

ProgIter is an alternative to *tqdm*. ProgIter implements much of the tqdm-API. The main difference between *ProgIter* and *tqdm* is that ProgIter does not use threading where as *tqdm* does.

Variables

- **iterable** (*iterable*) – An iterable iterable
- **desc** (*str*) – description label to show with progress
- **total** (*int*) – Maximum length of the process. If not specified, we estimate it from the iterable, if possible.
- **freq** (*int, default=1*) – How many iterations to wait between messages.
- **adjust** (*bool, default=True*) – if True freq is adjusted based on time_thresh
- **eta_window** (*int, default=64*) – number of previous measurements to use in eta calculation
- **clearline** (*bool, default=True*) – if True messages are printed on the same line otherwise each new progress message is printed on new line.
- **adjust** – if True *freq* is adjusted based on time_thresh
- **time_thresh** (*float, default=2.0*) – desired amount of time to wait between messages if adjust is True otherwise does nothing
- **show_times** (*bool, default=True*) – shows rate, eta, and wall (defaults to True)
- **initial** (*int, default=0*) – starting index offset (defaults to 0)
- **stream** (*file, default=sys.stdout*) – stream where progress information is written to
- **enabled** (*bool, default=True*) – if False nothing happens.
- **chunksize** (*int, optional*) – indicates that each iteration processes a batch of this size. Iteration rate is displayed in terms of single-items.
- **verbose** (*int*) – verbosity mode, which controls clearline, adjust, and enabled. The following maps the value of *verbose* to its effect. 0: enabled=False, 1: enabled=True with clearline=True and adjust=True, 2: enabled=True with clearline=False and adjust=True, 3: enabled=True with clearline=False and adjust=False

---

Note: Either use ProgIter in a with statement or call prog.end() at the end of the computation if there is a possibility that the entire iterable may not be exhausted.

Note: ProgIter is an alternative to *tqdm*. The main difference between *ProgIter* and *tqdm* is that ProgIter does not use threading where as *tqdm* does. *ProgIter* is simpler than *tqdm* and thus more stable in certain circumstances.

SeeAlso: tqdm - https://pypi.python.org/pypi/tqdm

References

http://datagenetics.com/blog/february12017/index.html

Example

```
>>> # doctest: +SKIP
>>> def is_prime(n):
...     return n >= 2 and not any(n % i == 0 for i in range(2, n))
>>> for n in ProgIter(range(100), verbose=1):
>>>     # do some work
>>>     is_prime(n)
100/100... rate=... Hz, total=..., wall=... EST
```

set_extra(*extra*)

specify a custom info appended to the end of the next message

---

Todo:

- [ ] extra is a bad name; come up with something better and rename

---

Example

```
>>> prog = ProgIter(range(100, 300, 100), show_times=False, verbose=3)
>>> for n in prog:
>>>     prog.set_extra('processesing num {}'.format(n))
0/2...
1/2...processesing num 100
2/2...processesing num 200
```

step(*inc=1*)

Manually step progress update, either directly or by an increment.

Parameters **inc** (*int, default=1*) – number of steps to increment

Example

```
>>> n = 3
>>> prog = ProgIter(desc='manual', total=n, verbose=3)
>>> # Need to manually begin and end in this mode
>>> prog.begin()
>>> for _ in range(n):
...     prog.step()
>>> prog.end()
```

Example

```
>>> n = 3
>>> # can be used as a context manager in manual mode
>>> with ProgIter(desc='manual', total=n, verbose=3) as prog:
```

(continues on next page)

(continued from previous page)

```
...     for _ in range(n):
...         prog.step()
```

begin()
Initializes information used to measure progress

This only needs to be used if this ProgIter is not wrapping an iterable. Does nothing if the this ProgIter is disabled.

end()
Signals that iteration has ended and displays the final message.

This only needs to be used if this ProgIter is not wrapping an iterable. Does nothing if the this ProgIter object is disabled or has already finished.

format_message()
builds a formatted progres message with the current values. This contains the special characters needed to clear lines.

Example

```
>>> self = ProgIter(clearline=False, show_times=False)
>>> print(repr(self.format_message()))
'    0/?... \n'
>>> self.begin()
>>> self.step()
>>> print(repr(self.format_message()))
' 1/?... \n'
```

Example

```
>>> self = ProgIter(chunksize=10, total=100, clearline=False,
>>>                 show_times=False, microseconds=True)
>>> # hack, microseconds=True for coverage, needs real test
>>> print(repr(self.format_message()))
' 0.00% of 10x100... \n'
>>> self.begin()
>>> self.update()  # tqdm alternative to step
>>> print(repr(self.format_message()))
' 1.00% of 10x100... \n'
```

ensure_newline()
use before any custom printing when using the progress iter to ensure your print statement starts on a new line instead of at the end of a progress line

Example

```
>>> # Unsafe version may write your message on the wrong line
>>> prog = ProgIter(range(4), show_times=False, verbose=1)
>>> for n in prog:
...     print('unsafe message')
 0/4...  unsafe message
```

(continues on next page)

(continued from previous page)

```
 1/4...  unsafe message
unsafe message
unsafe message
 4/4...
>>> # apparently the safe version does this too.
>>> print('---')
---
>>> prog = ProgIter(range(4), show_times=False, verbose=1)
>>> for n in prog:
...     prog.ensure_newline()
...     print('safe message')
 0/4...
safe message
 1/4...
safe message
safe message
safe message
 4/4...
```

display_message()
Writes current progress to the output stream

1.1.2 Module contents

ProgIter lets you measure and print the progress of an iterative process. This can be done either via an iterable interface or using the manual api. Using the iterable inferface is most common.

ProgIter was originally developed independantly of `tqdm`, but the newer versions of this library have been designed to be compatible with tqdm-API. `ProgIter` is now a (mostly) drop-in alternative to **tqdm_**. The `tqdm` library may be more appropriate in some cases. *The main advantage of ‘‘ProgIter‘‘ is that it does not use any python threading*, and therefore can be safer with code that makes heavy use of multiprocessing. **‘The reason‘_** for this is that threading before forking may cause locks to be duplicated across processes, which may lead to deadlocks.

ProgIter is simpler than tqdm, which may be desirable for some applications. However, this also means ProgIter is not as extensible as tqdm. If you want a pretty bar, use tqdm; if you want useful information (rate, fraction-complete, estimated time remaining, time taken so far, current wall time) about your iteration by default, use progiter.

Example

The basic usage of ProgIter is simple and intuitive. Just wrap a python iterable. The following example wraps a `range` iterable and prints reported progress to stdout as the iterable is consumed. The `ProgIter` object accepts various keyword arguments to modify the details of how progress is measured and reported. See API documentation of the `ProgIter` classs here: https://progiter.readthedocs.io/en/latest/progiter.progiter.html#progiter.progiter.ProgIter

Note that by default ProgIter reports information about iteration-rate, fraction-complete, estimated time remaining, time taken so far, and the current wall time.

```
>>> from progiter import ProgIter
>>> def is_prime(n):
...     return n >= 2 and not any(n % i == 0 for i in range(2, n))
>>> for n in ProgIter(range(1000), verbose=2):
>>>     # do some work
>>>     is_prime(n)
    0/1000... rate=0.00 Hz, eta=?, total=0:00:00, wall=14:05 EST
```

(continues on next page)

(continued from previous page)

```
   1/1000... rate=82241.25 Hz, eta=0:00:00, total=0:00:00, wall=14:05 EST
 257/1000... rate=177204.69 Hz, eta=0:00:00, total=0:00:00, wall=14:05 EST
 642/1000... rate=94099.22 Hz, eta=0:00:00, total=0:00:00, wall=14:05 EST
1000/1000... rate=71886.74 Hz, eta=0:00:00, total=0:00:00, wall=14:05 EST
```

class progiter.**ProgIter**(*iterable=None*, *desc=None*, *total=None*, *freq=1*, *initial=0*, *eta_window=64*, *clearline=True*, *adjust=True*, *time_thresh=2.0*, *show_times=True*, *enabled=True*, *verbose=None*, *stream=None*, *chunksize=None*, ***kwargs*)

Bases: progiter.progiter._TQDMCompat, progiter.progiter._BackwardsCompat

Prints progress as an iterator progresses

ProgIter is an alternative to *tqdm*. ProgIter implements much of the tqdm-API. The main difference between *ProgIter* and *tqdm* is that ProgIter does not use threading where as *tqdm* does.

Variables

- **iterable** (*iterable*) – An iterable iterable
- **desc** (*str*) – description label to show with progress
- **total** (*int*) – Maximum length of the process. If not specified, we estimate it from the iterable, if possible.
- **freq** (*int, default=1*) – How many iterations to wait between messages.
- **adjust** (*bool, default=True*) – if True freq is adjusted based on time_thresh
- **eta_window** (*int, default=64*) – number of previous measurements to use in eta calculation
- **clearline** (*bool, default=True*) – if True messages are printed on the same line otherwise each new progress message is printed on new line.
- **adjust** – if True *freq* is adjusted based on time_thresh
- **time_thresh** (*float, default=2.0*) – desired amount of time to wait between messages if adjust is True otherwise does nothing
- **show_times** (*bool, default=True*) – shows rate, eta, and wall (defaults to True)
- **initial** (*int, default=0*) – starting index offset (defaults to 0)
- **stream** (*file, default=sys.stdout*) – stream where progress information is written to
- **enabled** (*bool, default=True*) – if False nothing happens.
- **chunksize** (*int, optional*) – indicates that each iteration processes a batch of this size. Iteration rate is displayed in terms of single-items.
- **verbose** (*int*) – verbosity mode, which controls clearline, adjust, and enabled. The following maps the value of *verbose* to its effect. 0: enabled=False, 1: enabled=True with clearline=True and adjust=True, 2: enabled=True with clearline=False and adjust=True, 3: enabled=True with clearline=False and adjust=False

---

Note: Either use ProgIter in a with statement or call prog.end() at the end of the computation if there is a possibility that the entire iterable may not be exhausted.

Note: ProgIter is an alternative to *tqdm*. The main difference between *ProgIter* and *tqdm* is that ProgIter does not use threading where as *tqdm* does. *ProgIter* is simpler than *tqdm* and thus more stable in certain circumstances.

SeeAlso: tqdm - https://pypi.python.org/pypi/tqdm

References

http://datagenetics.com/blog/february12017/index.html

Example

```
>>> # doctest: +SKIP
>>> def is_prime(n):
...     return n >= 2 and not any(n % i == 0 for i in range(2, n))
>>> for n in ProgIter(range(100), verbose=1):
>>>     # do some work
>>>     is_prime(n)
100/100... rate=... Hz, total=..., wall=... EST
```

set_extra(*extra*)

specify a custom info appended to the end of the next message

Todo:

- [ ] extra is a bad name; come up with something better and rename

Example

```
>>> prog = ProgIter(range(100, 300, 100), show_times=False, verbose=3)
>>> for n in prog:
>>>     prog.set_extra('processesing num {}'.format(n))
0/2...
1/2...processesing num 100
2/2...processesing num 200
```

step(*inc=1*)

Manually step progress update, either directly or by an increment.

Parameters **inc** (*int, default=1*) – number of steps to increment

Example

```
>>> n = 3
>>> prog = ProgIter(desc='manual', total=n, verbose=3)
>>> # Need to manually begin and end in this mode
>>> prog.begin()
>>> for _ in range(n):
```

(continues on next page)

(continued from previous page)

```
...     prog.step()
>>> prog.end()
```

Example

```
>>> n = 3
>>> # can be used as a context manager in manual mode
>>> with ProgIter(desc='manual', total=n, verbose=3) as prog:
...     for _ in range(n):
...         prog.step()
```

begin()
Initializes information used to measure progress

This only needs to be used if this ProgIter is not wrapping an iterable. Does nothing if the this ProgIter is disabled.

end()
Signals that iteration has ended and displays the final message.

This only needs to be used if this ProgIter is not wrapping an iterable. Does nothing if the this ProgIter object is disabled or has already finished.

format_message()
builds a formatted progres message with the current values. This contains the special characters needed to clear lines.

Example

```
>>> self = ProgIter(clearline=False, show_times=False)
>>> print(repr(self.format_message()))
'    0/?... \n'
>>> self.begin()
>>> self.step()
>>> print(repr(self.format_message()))
' 1/?... \n'
```

Example

```
>>> self = ProgIter(chunksize=10, total=100, clearline=False,
>>>                 show_times=False, microseconds=True)
>>> # hack, microseconds=True for coverage, needs real test
>>> print(repr(self.format_message()))
' 0.00% of 10x100... \n'
>>> self.begin()
>>> self.update()  # tqdm alternative to step
>>> print(repr(self.format_message()))
' 1.00% of 10x100... \n'
```

ensure_newline()
use before any custom printing when using the progress iter to ensure your print statement starts on a new line instead of at the end of a progress line

Example

```
>>> # Unsafe version may write your message on the wrong line
>>> prog = ProgIter(range(4), show_times=False, verbose=1)
>>> for n in prog:
...     print('unsafe message')
 0/4...  unsafe message
 1/4...  unsafe message
unsafe message
unsafe message
 4/4...
>>> # apparently the safe version does this too.
>>> print('---')
---
>>> prog = ProgIter(range(4), show_times=False, verbose=1)
>>> for n in prog:
...     prog.ensure_newline()
...     print('safe message')
 0/4...
safe message
 1/4...
safe message
safe message
safe message
 4/4...
```

display_message()
Writes current progress to the output stream

Python Module Index

p

Index