progiter Documentation

Jon Crall

Jun 20, 2023

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 interface is most common.

ProgIter is *unthreaded*. This differentiates it from tqdm and rich.progress which use a *threaded* implementation. The choice of implementation has different tradeoffs and neither is strictly better than the other. An unthreaded progress bar provides synchronous uncluttered logging, increased stability, and — unintuitively —- speed (due to Python's GIL). Meanwhile threaded progress bars are more responsive, able to update multiple stdout lines at a time, and can look prettier (unless you try to log stdout to disk).

ProgIter was originally developed independently 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 (or rich); if you want useful information about your iteration by default, use progiter.

Package level documentation can be found at: https://progiter.readthedocs.io/en/latest/

Example

The basic usage of ProgIter is simple and intuitive: wrap a python iterable. The following example wraps a `range` iterable and reports 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` class here: https://progiter.readthedocs.io/en/latest/progiter.progiter.html#progiter.progiter.ProgIter

```
>>> 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)
0.00%    0/1000... rate=0 Hz, eta=?, total=0:00:00
0.60%    6/1000... rate=76995.12 Hz, eta=0:00:00, total=0:00:00
100.00% 1000/1000... rate=266488.22 Hz, eta=0:00:00, total=0:00:00
```

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 interface is most common.

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=114326.51 Hz, eta=0:00:00, total=0:00:00
```

For more complex applications is may sometimes be desirable 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
manual 1/3... rate=14454.63 Hz, eta=0:00:00, total=0:00:00
manual 2/3... rate=17485.42 Hz, eta=0:00:00, total=0:00:00
manual 3/3... rate=21689.78 Hz, eta=0:00:00, total=0:00:00
```

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, show_wall=True)
>>> for n in prog:
>>>     if n == 97:
>>>         print('!!! Special print at n=97 !!!')
>>>     if is_prime(n):
```

(continues on next page)

(continued from previous page)

```
>>>         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=2020-10-23 17:27 EST
check primes    1/1000... rate=95547.49 Hz, eta=0:00:00, total=0:00:00, wall=2020-10-23␣
↪17:27 EST
check primes    4/1000...Biggest prime so far: 3 rate=41062.28 Hz, eta=0:00:00,␣
↪total=0:00:00, wall=2020-10-23 17:27 EST
check primes   16/1000...Biggest prime so far: 13 rate=85340.61 Hz, eta=0:00:00,␣
↪total=0:00:00, wall=2020-10-23 17:27 EST
check primes   64/1000...Biggest prime so far: 61 rate=164739.98 Hz, eta=0:00:00,␣
↪total=0:00:00, wall=2020-10-23 17:27 EST
!!! Special print at n=97 !!!
check primes  256/1000...Biggest prime so far: 251 rate=206287.91 Hz, eta=0:00:00,␣
↪total=0:00:00, wall=2020-10-23 17:27 EST
check primes  512/1000...Biggest prime so far: 509 rate=165271.92 Hz, eta=0:00:00,␣
↪total=0:00:00, wall=2020-10-23 17:27 EST
check primes  768/1000...Biggest prime so far: 761 rate=136480.12 Hz, eta=0:00:00,␣
↪total=0:00:00, wall=2020-10-23 17:27 EST
check primes 1000/1000...Biggest prime so far: 997 rate=115214.95 Hz, eta=0:00:00,␣
↪total=0:00:00, wall=2020-10-23 17:27 EST
```

CHAPTER

ONE

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 interface is most common.

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=114326.51 Hz, eta=0:00:00, total=0:00:00
```

For more complex applications is may sometimes be desirable 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
manual 1/3... rate=14454.63 Hz, eta=0:00:00, total=0:00:00
manual 2/3... rate=17485.42 Hz, eta=0:00:00, total=0:00:00
manual 3/3... rate=21689.78 Hz, eta=0:00:00, total=0:00:00
```

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, show_wall=True)
>>> 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=2020-10-23 17:27 EST
check primes    1/1000... rate=95547.49 Hz, eta=0:00:00, total=0:00:00, wall=2020-10-23␣
↪17:27 EST
check primes    4/1000...Biggest prime so far: 3 rate=41062.28 Hz, eta=0:00:00,␣
↪total=0:00:00, wall=2020-10-23 17:27 EST
check primes   16/1000...Biggest prime so far: 13 rate=85340.61 Hz, eta=0:00:00,␣
↪total=0:00:00, wall=2020-10-23 17:27 EST
check primes   64/1000...Biggest prime so far: 61 rate=164739.98 Hz, eta=0:00:00,␣
↪total=0:00:00, wall=2020-10-23 17:27 EST
!!! Special print at n=97 !!!
check primes  256/1000...Biggest prime so far: 251 rate=206287.91 Hz, eta=0:00:00,␣
↪total=0:00:00, wall=2020-10-23 17:27 EST
check primes  512/1000...Biggest prime so far: 509 rate=165271.92 Hz, eta=0:00:00,␣
↪total=0:00:00, wall=2020-10-23 17:27 EST
check primes  768/1000...Biggest prime so far: 761 rate=136480.12 Hz, eta=0:00:00,␣
↪total=0:00:00, wall=2020-10-23 17:27 EST
check primes 1000/1000...Biggest prime so far: 997 rate=115214.95 Hz, eta=0:00:00,␣
↪total=0:00:00, wall=2020-10-23 17:27 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_percent=True*, *show_times=True*, *show_rate=True*, *show_eta=True*, *show_total=True*, *show_wall=False*, *enabled=True*, *verbose=None*, *stream=None*, *chunksize=None*, *rel_adjust_limit=4.0*, *homogeneous='auto'*, *timer=None*, ***kwargs*)

Bases: _TQDMCompat, _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 whereas *tqdm* does.

Variables

- **iterable** (*List | Iterable*) – A list or iterable to loop over
- **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*) – How many iterations to wait between messages. Defaults to 1.
- **eta_window** (*int*) – number of previous measurements to use in eta calculation, default=64
- **clearline** (*bool*) – if True messages are printed on the same line otherwise each new progress message is printed on new line. default=True
- **adjust** (*bool*) – if True *freq* is adjusted based on time_thresh. This may be overwritten depending on the setting of verbose. default=True
- **time_thresh** (*float*) – desired amount of time to wait between messages if adjust is True otherwise does nothing, default=2.0
- **show_percent** (*bool*) – if True show percent progress. Default=True
- **show_times** (*bool*) – if False do not show rate, eta, or wall time. default=True Deprecated. Use show_rate / show_eta / show_wall instead.
- **show_rate** (*bool*) – show / hide rate, default=True
- **show_eta** (*bool*) – show / hide estimated time of arival (i.e. time to completion), default=True
- **show_wall** (*bool*) – show / hide wall time, default=False
- **initial** (*int*) – starting index offset, default=0
- **stream** (*IO*) – stream where progress information is written to, default=sys.stdout
- **timer** (*callable*) – the timer object to use. Defaults to time.perf_counter().
- **enabled** (*bool*) – if False nothing happens. default=True
- **chunksize** (*int | None*) – indicates that each iteration processes a batch of this size. Iteration rate is displayed in terms of single-items.
- **rel_adjust_limit** (*float*) – Maximum factor update frequency can be adjusted by in a single step. default=4.0
- **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

- **homogeneous** (*bool | str*) – Indicate if the iterable is likely to take a uniform or homogeneous amount of time per iteration. When True we can enable a speed optimization. When False, the time estimates are more accurate. Default to "auto", which attempts to determine if it is safe to use True. Has no effect if `adjust` is 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 whereas *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

```
>>>
>>> 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, show_wall=True):
>>>     # do some work
>>>     is_prime(n)
100/100... rate=... Hz, total=..., wall=...
```

set_extra(*extra*)

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

Parameters
extra (*str | Callable*) – a constant or dynamically constructed extra 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))
 0.00% 0/2...
 50.00% 1/2...processesing num 100
 100.00% 2/2...processesing num 200
```

begin()

Initializes information used to measure progress

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

Returns
a chainable self-reference

Return type
ProgIter

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 this ProgIter object is disabled or has already finished.

step(*inc=1*, *force=False*)

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

Parameters

- **inc** (*int, default=1*) – number of steps to increment
- **force** (*bool, default=False*) – if True forces progress display

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

format_message()

Exists only for backwards compatibility.

See *format_message_parts* for more recent API.

format_message_parts()

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_parts()[1]))
'    0/?... '
>>> self.begin()
>>> self.step()
>>> print(repr(self.format_message_parts()[1]))
' 1/?... '
```

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_parts()[1]))
' 0.00% of 10x100... '
>>> self.begin()
>>> self.update()  # tqdm alternative to step
>>> print(repr(self.format_message_parts()[1]))
' 1.00% of 10x100... '
```

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(3), show_times=False, freq=2, adjust=False,
...                 time_thresh=0)
>>> for n in prog:
...     print('unsafe message')
 0.00% 0/3... unsafe message
unsafe message
 66.67% 2/3... unsafe message
 100.00% 3/3...
>>> # apparently the safe version does this too.
>>> print('---')
---
>>> prog = ProgIter(range(3), show_times=False, freq=2, adjust=False,
...                 time_thresh=0)
>>> for n in prog:
...     prog.ensure_newline()
...     print('safe message')
 0.00% 0/3...
safe message
safe message
 66.67% 2/3...
safe message
 100.00% 3/3...
```

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 interface is most common.

ProgIter is *unthreaded*. This differentiates it from tqdm and rich.progress which use a *threaded* implementation. The choice of implementation has different tradeoffs and neither is strictly better than the other. An unthreaded progress bar provides synchronous uncluttered logging, increased stability, and — unintuitively —- speed (due to Python's GIL). Meanwhile threaded progress bars are more responsive, able to update multiple stdout lines at a time, and can look prettier (unless you try to log stdout to disk).

ProgIter was originally developed independently 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 (or rich); if you want useful information about your iteration by default, use progiter.

Package level documentation can be found at: https://progiter.readthedocs.io/en/latest/

Example

The basic usage of ProgIter is simple and intuitive: wrap a python iterable. The following example wraps a `range` iterable and reports 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` class here: https://progiter.readthedocs.io/en/latest/progiter.progiter.html#progiter.progiter.ProgIter

```
>>> 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)
0.00%    0/1000... rate=0 Hz, eta=?, total=0:00:00
0.60%    6/1000... rate=76995.12 Hz, eta=0:00:00, total=0:00:00
100.00% 1000/1000... rate=266488.22 Hz, eta=0:00:00, total=0:00:00
```

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_percent=True*, *show_times=True*, *show_rate=True*, *show_eta=True*, *show_total=True*, *show_wall=False*, *enabled=True*, *verbose=None*, *stream=None*, *chunksize=None*, *rel_adjust_limit=4.0*, *homogeneous='auto'*, *timer=None*, ***kwargs*)

Bases: `_TQDMCompat`, `_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 whereas *tqdm* does.

Variables

- **iterable** (*List | Iterable*) – A list or iterable to loop over
- **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*) – How many iterations to wait between messages. Defaults to 1.
- **eta_window** (*int*) – number of previous measurements to use in eta calculation, default=64
- **clearline** (*bool*) – if True messages are printed on the same line otherwise each new progress message is printed on new line. default=True
- **adjust** (*bool*) – if True *freq* is adjusted based on time_thresh. This may be overwritten depending on the setting of verbose. default=True
- **time_thresh** (*float*) – desired amount of time to wait between messages if adjust is True otherwise does nothing, default=2.0
- **show_percent** (*bool*) – if True show percent progress. Default=True
- **show_times** (*bool*) – if False do not show rate, eta, or wall time. default=True Deprecated. Use show_rate / show_eta / show_wall instead.
- **show_rate** (*bool*) – show / hide rate, default=True
- **show_eta** (*bool*) – show / hide estimated time of arival (i.e. time to completion), default=True
- **show_wall** (*bool*) – show / hide wall time, default=False
- **initial** (*int*) – starting index offset, default=0
- **stream** (*IO*) – stream where progress information is written to, default=sys.stdout
- **timer** (*callable*) – the timer object to use. Defaults to `time.perf_counter()`.
- **enabled** (*bool*) – if False nothing happens. default=True
- **chunksize** (*int | None*) – indicates that each iteration processes a batch of this size. Iteration rate is displayed in terms of single-items.
- **rel_adjust_limit** (*float*) – Maximum factor update frequency can be adjusted by in a single step. default=4.0
- **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
- **homogeneous** (*bool | str*) – Indicate if the iterable is likely to take a uniform or homogeneous amount of time per iteration. When True we can enable a speed optimization. When False, the time estimates are more accurate. Default to "auto", which attempts to determine if it is safe to use True. Has no effect if `adjust` is 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 whereas *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

```
>>>
>>> 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, show_wall=True):
>>>     # do some work
>>>     is_prime(n)
100/100... rate=... Hz, total=..., wall=...
```

set_extra(*extra*)

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

Parameters
extra (*str | Callable*) – a constant or dynamically constructed extra 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))
 0.00% 0/2...
 50.00% 1/2...processesing num 100
 100.00% 2/2...processesing num 200
```

begin()

Initializes information used to measure progress

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

Returns
a chainable self-reference

Return type
ProgIter

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 this ProgIter object is disabled or has already finished.

step(*inc=1*, *force=False*)

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

Parameters

- **inc** (*int, default=1*) – number of steps to increment
- **force** (*bool, default=False*) – if True forces progress display

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

format_message()

Exists only for backwards compatibility.

See *format_message_parts* for more recent API.

format_message_parts()

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_parts()[1]))
'    0/?... '
>>> self.begin()
>>> self.step()
>>> print(repr(self.format_message_parts()[1]))
' 1/?... '
```

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_parts()[1]))
' 0.00% of 10x100... '
>>> self.begin()
>>> self.update()  # tqdm alternative to step
>>> print(repr(self.format_message_parts()[1]))
' 1.00% of 10x100... '
```

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(3), show_times=False, freq=2, adjust=False,
...                 time_thresh=0)
>>> for n in prog:
...     print('unsafe message')
 0.00% 0/3... unsafe message
unsafe message
 66.67% 2/3... unsafe message
 100.00% 3/3...
>>> # apparently the safe version does this too.
>>> print('---')
---
>>> prog = ProgIter(range(3), show_times=False, freq=2, adjust=False,
...                 time_thresh=0)
>>> for n in prog:
...     prog.ensure_newline()
...     print('safe message')
 0.00% 0/3...
safe message
safe message
 66.67% 2/3...
safe message
 100.00% 3/3...
```

display_message()

Writes current progress to the output stream

class progiter.**ProgressManager**(*backend='rich'*, ***kwargs*)

Bases: BaseProgIterManager

A progress manager.

Manage multiple progress bars, either with rich or ProgIter.

CommandLine:

xdoctest -m progiter.manager ProgressManager:0 xdoctest -m progiter.manager ProgressManager:1

xdoctest -m progiter.manager ProgressManager:2

Example

```
>>> from progiter.manager import ProgressManager
>>> from progiter import progiter
>>> # Can use plain progiter or rich
>>> # The usecase for plain progiter is when threads / live output
>>> # is not desirable and you just want plain stdout progress
>>> pman = ProgressManager(backend='progiter')
>>> with pman:
>>>     oprog = pman.progiter(range(20), desc='outer loop', verbose=3)
>>>     for i in oprog:
>>>         oprog.set_postfix(f'Doing step {i}', refresh=False)
>>>         for i in pman.progiter(range(100), desc=f'inner loop {i}'):
>>>             pass
>>> #
>>> # xdoctest: +REQUIRES(module:rich)
>>> self = pman = ProgressManager(backend='rich')
>>> pman = ProgressManager(backend='rich')
>>> with pman:
>>>     oprog = pman.progiter(range(20), desc='outer loop', verbose=3)
>>>     for i in oprog:
>>>         oprog.set_postfix(f'Doing step {i}', refresh=False)
>>>         for i in pman.progiter(range(100), desc=f'inner loop {i}'):
>>>             pass
```

Example

```
>>> # A fairly complex example
>>> # xdoctest: +REQUIRES(module:rich)
>>> from progiter.manager import ProgressManager
>>> import time
>>> delay = 0.00005
>>> N_inner = 300
>>> N_outer = 11
>>> self = pman = ProgressManager(backend='rich')
>>> with pman:
>>>     oprog = pman(range(N_outer), desc='outer loop')
>>>     for i in oprog:
>>>         if i > 7:
>>>             self.update_info(f'The info panel gives detailed updates\nWe are␣
↪now at step {i}\nWe are just about done now')
>>>         elif i > 5:
>>>             self.update_info(f'The info panel gives detailed updates\nWe are␣
↪now at step {i}')
>>>         oprog.set_postfix(f'Doing step {i}')
>>>         N = 1000
>>>         for j in pman(iter(range(N_inner)), total=None if i % 2 == 0 else N_
↪inner, desc=f'inner loop {i}', transient=i < 4):
>>>             time.sleep(delay)
```

Example

```
>>> # Test complex example over a grid of parameters
>>> # xdoctest: +REQUIRES(module:ubelt)
>>> # xdoctest: +REQUIRES(module:rich)
>>> import ubelt as ub
>>> from progiter.manager import ProgressManager, ManagedProgIter
>>> import time
>>> delay = 0.000005
>>> N_inner = 300
>>> N_outer = 11
>>> basis = {
>>>     'with_info': [0, 1],
>>>     'backend': ['progiter', 'rich'],
>>>     'enabled': [0, 1],
>>>     #'with_info': [1],
>>> }
>>> grid = list(ub.named_product(basis))
>>> grid_prog = ManagedProgIter(grid, desc='Test cases over grid', verbose=3)
>>> grid_prog.update_info('Here we go')
>>> for item in grid:
>>>     grid_prog.ensure_newline()
>>>     grid_prog.update_info(f'Running grid test {ub.urepr(item, nl=1)}')
>>>     print('\n\n')
>>>     self = ProgressManager(backend=item['backend'], enabled=item['enabled'])
>>>     with self:
>>>         outer_prog = self.progiter(range(N_outer), desc='outer loop')
>>>         for i in outer_prog:
>>>             if item['with_info']:
>>>                 if i > 7:
>>>                     outer_prog.update_info(f'The info panel gives detailed␣
↪updates\nWe are now at step {i}\nWe are just about done now')
>>>                 elif i > 5:
>>>                     outer_prog.update_info(f'The info panel gives detailed␣
↪updates\nWe are now at step {i}')
>>>             outer_prog.set_postfix(f'Doing step {i}')
>>>             inner_kwargs = dict(
>>>                 total=None if i % 2 == 0 else N_inner,
>>>                 transient=i < 4,
>>>                 time_thresh=delay * 2.3,
>>>                 desc=f'inner loop {i}',
>>>             )
>>>             for j in self.progiter(iter(range(N_inner)), **inner_kwargs):
>>>                 time.sleep(delay)
>>>     grid_prog.update_info(f'Finished test item')
```

Example

```
>>> # Demo manual usage
>>> # xdoctest: +REQUIRES(module:rich)
>>> from progiter.manager import ProgressManager
>>> from progiter import manager
>>> import time
>>> pman = ProgressManager()
>>> pman.start()
>>> task1 = pman.progiter(desc='task1', total=100)
>>> task2 = pman.progiter(desc='task2')
>>> for i in range(100):
>>>     task1.update()
>>>     task2.update(2)
>>>     time.sleep(0.001)
>>> ProgressManager.stopall()
```

Example

```
>>> # Demo manual usage (progiter backend)
>>> from progiter.manager import ProgressManager
>>> from progiter import manager
>>> import time
>>> pman = ProgressManager(backend='progiter', adjust=0, freq=1)
>>> pman.start()
>>> task1 = pman.progiter(desc='task1', total=12)
>>> task2 = pman.progiter(desc='task2')
>>> task1.update()
>>> task2.update()
>>> for i in range(10):
>>>     time.sleep(0.001)
>>>     task1.update()
>>>     time.sleep(0.001)
>>>     task2.update(2)
>>> ProgressManager.stopall()
```

progiter(**args*, ***kw*)

update_info(*text*)

start()

stop(**args*, ***kwargs*)

classmethod stopall()

Stop all background progress threads (likely only 1 exists)

Ignore:

from progiter import manager manager.ProgressManager.stopall()

PYTHON MODULE INDEX

p

INDEX