Pyux - Tools to ease tracking scripts

Release 0.1.4

Apr 14, 2020

Documentation

Pyux is a set of functions and classes that help with keeping track of what is happening during a script execution. It contains simple but helpful classes. Contributions *to the package are welcomed !*

Three modules are available :

- `console` provide tools to print stuff to console while a script is executing
- `time` provide tools in relation with time, which is measuring it or waiting for it
- `logging` contains a wrapper around a logger from `logging` module to spare you configuring one.

All classes and functions can be imported straight from `pyux`, without needing to pass the module name.

CHAPTER 1

Installation

You can download `pyux` from PyPi. There is only one requirement which is package colorama, which will be automatically installed with `pyux`.

```
pip install pyux-track
```

CHAPTER 2

Demonstration script

Once installed, the package comes with a demonstration script that you can use to see how the available classes behave. The demo is exhaustive and interactive so you can skip some sections if you want, or quit it if you get bored. Launch the demo by typing in a terminal :

```
python -m pyux
```

Please note that the demo does not provide any information on how to use the classes : you will find detailed explanations in the documentation.

2.1 Console

This module contains classes that print things to the terminal during execution. Most of them are meant to be used as decorators of a for statement, where they can help track where you are in the loop.

You can also use them manually to customise behaviour within a loop or anywhere else in your scripts.

2.1.1 Wheel

Use `Wheel` as an iterable's decoration to have a turning *wheel* printed during a `for` loop, turning as fast as the iterations go. By default the iteration index is printed next to the wheel, but you can choose to print the iteration value instead of the iteration index with argument `print_value = True`.

When used over a standard `range(n)`, it accepts `n` as an argument for the iterable.

```
from pyux.console import Wheel
from time import sleep

myrange = 'An iterable string'
for step in Wheel(myrange, print_value = True):
    sleep(0.2)
```

(continues on next page)

(continued from previous page)

```
for step in Wheel(20):
   sleep(0.1)
```

If you must, you can also instantiate `Wheel` outside of a loop then trigger it manually. It is useful mainly to customise the message you want to print.

Note that when using it manually, you may want to *close* the wheel when you're done : it prints the length of the iterable or a message if provided, and flushes a new line. Otherwise the console cursor is still on the previous line. When decorating a for loop, it is automatically called when the loop finishes.

```
from time import sleep
from pyux.console import Wheel

wheel = Wheel()                  # no need for an iterator argument in that case
for index, word in enumerate(['coffee', 'vanilla', 'cocoa']):
    wheel.print(step = index, message = 'Running on word %s              ' % word)
    sleep(1)                    # renders best if iterations are not too fast
wheel.close(message = 'No more words to think about.')
```

`Wheel` can also be used to decorate a generator object rather than an iterable, which is especially useful with `os.walk()`, for instance, to know both if your script is still running and the total number of read paths. This also makes it compatible with `enumerate()`.

2.1.2 Speak and wordy

`wordy` is a decorator function that prints a message at each call of the decorated function. You can catch exceptions from the decorated function with `catch = True` and print a specific message when the exception is catch.

By default, `.` is printed if the call is successful, and `!` if it isn't, allowing easy and concise error tracking within loops. When catching an exception, the exception is returned in place of the function response.

```
from pyux.console import wordy

@wordy(catch = True, failure_message = 'x')
def error_4(x):
    if x == 4:
        raise ValueError
[error_4(x) for x in range(10)]
```

Class `Speak` does roughly the same as `wordy`, but decorates an iterable in a for loop rather than a function. It thus prints a message at each iteration or every given number of iterations. It does not provide any exception catching environment. For that, you'll have to decorate the function which you want to catch from with `wordy` instead of using `Speak`.

As for all other iterable's decorators in this package, standard `range(n)` iterables can be given with just `n` as the `iterable` argument.

```
from pyux.console import Speak

for x in Speak(45, every = 5):
    pass
```

Both `wordy` and `Speak` print their messages with `sys.stdout.write()`, and not `print()`, so that the console cursor can stay on the same line within the same loop. When using `Speak` on an iterable in a for loop, a `close()` method is automatically called when the loop finished to flush a new line. When used *manually*, you'll have to flush `\n` (or `print('')`) to make the cursor go to the new line.

2.1.3 ProgressBar

Use `ProgressBar` as an iteratable's decoration to have a progress bar printed during a `for` loop. It mimics, in a extremely simplified way, the behavior of the great tqdm progress bar. As for `Wheel`, you can also use it manually.

Integers for `iterable` argument are read as `range(n)`.

```
from pyux.console import ProgressBar
from time import sleep

for index in ProgressBar(2500):
    sleep(0.001)
```

Since `ProgressBar` needs to know at initialisation the total number of iterations (to calculate the bar's width and the percentage), it is not usable with generators. A workaround is to give it an approximate number of iterations as `iterable` argument, and use it manually. Careful though, should the approximation be too short, the bar will expand further than the console width (or never reach 100% if too big).

```
from pyux.console import ProgressBar
from time import sleep

def simple_generator(n):
   for i in range(n):
      yield i

bar = ProgressBar(1000)
for value in simple_generator(1200):
   sleep(0.002)
   bar.print(step = value)
print('Too long !')

bar.current_console_step = 0    # resetting bar to zero
for value in simple_generator(800):
    sleep(0.002)
    bar.print(step = value)
print('Too short !')
```

A *manual* ProgressBar can also be used to track progression on scripts with distinct stages that are not necessarily in the form of a loop, should there be so many of them that it makes sense.

Do use `ProgressBar.close()` method to be sure that the 100% iteration will be printed and the console cursor flushed to a new line when you use it manually. When decorating a for statement, it is automatically called when the loop finishes.

```
bar = ProgressBar()

# here goes stage 1
bar.print(step = 1)

# here goes ...
bar.print(step = ...)

bar.close()
```

2.1.4 ColourPen

Use `ColourPen` as its name indicates : to write colored text in terminal. It uses `colorama` package. Use a single instance for different styles thanks to the possibility of chaining `write` instructions.

```
from pyux.console import ColourPen

pen = ColourPen()
pen.write('Hello', color = 'cyan', style = 'normal')\
    .write(' this is another', color = 'red')\
    .write(' color on the same line.', color = 'green', newline = True)\
    .write("The same color on a new line. I'm reseting styles after that.", newline =␣
↪True, reset = True)\
    .write('A normal goodbye.')\
    .close()
```

`ColourPen.close()` resets styles to normal, flushes to a new line and closes `colorama`, which means that if you do not initialise a pen instance again, the colouring and styling won't work anymore.

2.1.5 Detailed documentation

class pyux.console.**ColourPen**

Bases: `object`

Write colored and styled messages in console.

Only one pen instance is needed to print texts with different styles. By default, styles and colours remain the same until changed or reset, allowing to write successive prints with the same format, without having to specify format at each call.

Available colors are :

- black
- red
- green
- yellow
- blue
- magenta
- cyan
- white
- reset (to reset color to default)

Available styles are :

- dim (thin)
- normal
- bright
- reset_all (to reset both color and style)

Any other value will raise a `ColorValueError` or a `StyleValueError`.

Example

```
>>> pen = ColourPen()
>>> pen.write('A blue bright message', color = 'cyan', style = 'bright')
>>> pen.write('Still a blue message', reset = True, newline = True)
>>> pen.write('That message is back to normal')
>>> pen.close()
```

close() → None

Reset all styles and close `colorama` util.

The method flushes an empty message with reset styles and a new line. Closing make the `sys.stdout` go back to normal : styling and colouring will not be recognised after it.

write(*message: str = ''*, *color: str = None*, *style: str = None*, *flush: bool = True*, *newline: bool = False*, *reset: bool = False*)

Write a colored and styled message.

The method returns `self` so that calls to the method can be chained from a single instance.

Parameters

- **message** (*str*) – default `''` : the message to write
- **color** (*str*) – default `None` : the colour to use, checked against possible values
- **style** (*str*) – default `None` : the style to use, checked against possible values
- **flush** (*bool*) – default `True` : flush the message to the console
- **newline** (*bool*) – default `False` : insert a new line after the message
- **reset** (*bool*) – default `False` : reset style and colour after writing the message

The default behavior is to pass on current style to subsequent calls, so you only need to style and colour once if you want to keep the same format for following messages. This is accomplished through `None` values given for the `color` and/or `style` arguments, which means that giving `None` does not ignore style and color !

To remove style for the message to print, use `style = 'RESET_ALL'`. To remove style *after* printing the styled message, use `reset = True`.

Message is printed with `sys.stdout.write()` rather than `print()`, allowing precise control over where to print in the console. The default behavior is to flush at each print and to keep the console cursor where it is.

You can decide to "write" several messages and flush them all at once, and you can add a newline after a message to get the equivalent of a `print()` statement.

Returns `self`

Raises `ColorValueError`, `StyleValueError`

Example

```
>>> pen = ColourPen()
>>> pen.write('Hello... ', color = 'cyan')        >>>      .write('Goodbye !',␣
↪reset = True, newline = True)
```

class pyux.console.**ProgressBar**(*iterable=None*, *ascii_only: bool = False*)

Bases: `object`

Print a progress bar with progression percentage and number of iterations.

`ProgressBar` is meant to be used as a decorator in a for statement : for each iteration, it prints the percentage of progress, the number of iterations against the total number of iterations to do, and a progress bar.

Parameters

- **iterable** (*iterable or int*) – default `None` : an iterable, or an integer for standard `range(n)` values
- **ascii_only** (*bool*) – Use ascii character = to print the bar

A progress bar can also be created manually if you want to call it outside of a for statement. However it still *needs* an iterable to know the total number of iterations. This means it cannot be used with generators.

A workaround if you really want to use it even if you don't know how long the iteration will last, give an approximation as an integer to the `iterable` argument. This might cause unwanted behaviour in console, especially if your approximation is too short.

Raise `TypeError` if `iterable` is not an iterable

Example

```
>>> from time import sleep
>>> for _ in ProgressBar(3000, ascii_only = True):
>>>     sleep(0.001)
```

close() → None
Print the bar with 100% completion.

When the for loop has finished, this prints a last progress bar with 100% completion. This method is automatically called at the end of a for loop when `ProgressBar` decorates a for statement.

Without closing, the iteration index would end just before reaching 100% due to index starting from zero rather than one.

Example

```
>>> ProgressBar(iterable = 'A string iterable').close()
```

print(*step: int*) → None
Print the progress bar for the current iteration.

class pyux.console.**Speak**(*iterable=None*, *every: int = 1*, *message: str = '.'*, *newline: bool = False*)
Bases: `object`

Print a message after each iteration or every *n* iterations.

This class decorates an iterable in a for statement by printing a message either after each iteration, or after a given amount of iterations.

Parameters

- **iterable** (*iterable or int*) – default `None` : an iterable, or an integer for standard `range(n)` values
- **every** (*int*) – default `1` : frequency of printing messages
- **message** (*str*) – default `'.'` : message to print
- **newline** (*bool*) – default `False` : flushes a new line after printing

Printing is done with `sys.stdout`, so that the default behavior is to keep the console cursor on the current line. To print the message on a newline at each step, use `newline = True`.

Example

```
>>> for _ in Speak(50, every = 5, message = '*'):
>>>     pass
```

close() → None
Flush a new line when `self.newline = False`.

This method is automatically called when the for loop finishes, so that the console cursor starts a new line when the loop is over.

Returns `None`

class pyux.console.**Wheel**(*iterable=None*, *print_value: bool = False*)
Bases: `object`

Print a wheel turning at the same speed as iterations.

This class decorates an iterable in a for statement by printing a turning wheel and the iteration index at each step.

Parameters

- **iterable** (*iterable or int*) – default `None` : an iterable, or an integer for standard `range(n)` values
- **print_value** (*bool*) – default `False` : print the iteration value instead of the iteration index next to the wheel

`Wheel` can also be used manually (without iterable), which is useful if you need to print messages different than the iteration value or index. It is compatible with generators, which makes it a good indicator that the code is still running when you do not know the total number of iterations beforehand.

Raise `TypeError` if `iterable` is not an iterable

Example

```
>>> for _ in Wheel(['a', 'b', 'c', 'd']):     # print the index
>>> for _ in Wheel(5):                        # same behaviour than previous
>>> for letter in Wheel(['a', 'b', 'c', 'd'], print_value = True):
>>>     # this whill print the iteration value (letters)
>>>     # rather than the iteration index
```

close(*message: str = None*) → None
Print the wheel for the last iteration then flush console.

This method makes the last printed value equal to the number of iterations. If a message is provided, it is printed instead of the former.

Parameters **message** (*str*) – default `None` : message to print next to the wheel

This method is automatically called when the wheel is used as decorator of an iterable in a for statement.

When `close` is called with no iterable given as argument at class instanciation, it will print `0` since a missing iterable as argument is internally seen as an iterable of length zero. A workaround is to use `close(message = '')` if you want the last printed value to stay, or any other message if you want it to be replaced by a closing message.

Example

```
>>> wheel = Wheel(iterable = 'A string iterable')
>>> wheel.close()
>>> wheel.close(message = 'Overriding length of iterable')
```

static print(*step: int*, *message: str = None*) → None
Print the wheel for a given step and message.

pyux.console.**wordy**(*message: str = '.', catch: bool = False, failure_message: str = '!', colors: tuple = None*)

Print a message at each function call.

A decorator to print a message at each function call with `sys.stdout`. Exceptions can be catch and returned as the function's result, or raised. If an exception is catch, a specific message can be printed.

Parameters

- **message** (*str*) – default `'.'` : message to print *after* function call
- **catch** (*bool*) – default `False` : catch an exception from the decorated function rather than raising it
- **failure_message** (*str*) – default `'!'` : message to print if an exception is catch during function call
- **colors** (*tuple*) – default `None` : a tuple of one or two colors for the the printed messages

Returns a function decorated with a printed message after execution

Both `message` and `failure_message` can be printed (internally using `ColourPen`). If `colors` is not specified, standard terminal tex color will be used. When `catch = True`, if specified, the `colors` tuple must be of length two.

Example

```
>>> @wordy()
>>> def one():
>>>     return 1
>>> [one() for _ in range(10)]
```

2.2 Time

This module contains classes that handle time : either measure it, wait or even control it !

2.2.1 Timer

`Timer` pause your script for the given number of seconds. With quite the same design as wheel, you may add a message next to the timer.

```
from pyux.time import Timer

# A timer for 3 seconds with a message
Timer(delay = 3, message = 'Pausing 3 secs, tired')

# A timer with no message
Timer(delay = 3)
```

The timer can also be used as an iterable's decoration within a for statement, when you repeatedly have to await the same delay at each iteration. Specifying `overwrite = True` allows each iteration to be rewritten on the same line, which is advised when used in that case.

Note that the first argument to `Timer` is `iterable` and not `delay`, and all of them have default values, so `Timer(5)` won't have the expected behaviour !

By default, a timer in a for loop prints the iteration index next to the timer. Use `pattern` argument to specify a prefix to add to the default iteration index (default to `''`), or `print_value` to print the iteration value rather than the index.

```
from pyux.time import Timer

for fruit in Timer(['banana', 'apple', 'tomato'], delay = 3, print_value = True):
   pass
```

Again, the `Timer.close()` makes the counter go to zero *included* and flushes a new line. It is called automatically when used as a loop decoration.

2.2.2 Wait

Use `wait` to decorate a function that you want to pause for a given amount of time before or after each execution. It can be useful for API calls in loops where you have await a certain time between each API call. The pause can be made before or after the function call, and `Timer` can be used instead of the built-in `sleep` function with `timer = True`.

```
from pyux.time import wait

@wait(delay = 3, before = True)
def do_nothing():
   return
do_nothing()
```

2.2.3 timeout

`timeout` decorates a function that you want to timeout after a given delay.

```
from pyux.time import timeout

@timeout(delay = 5)
def unstopable_function():
   i = 0
   while i >= 0:
      i += 1
   return i

unstopable_function()     # timeout stops execution after 5 seconds
```

2.2.4 Chronos

Use `Chronos` to measure user execution time, for a script or a loop. It works as a stopwatch : rather than wrapping around and timing an expression, it triggers at start, then the method `Chronos.lap()` records time with `timeit.default_timer()` each time it is called (thus resembling a lap button on a stopwatch).

```
from time import sleep
from pyux.time import Chronos

chrono = Chronos()
print('Step one with duration approx. 1.512 secs')
```

(continues on next page)

(continued from previous page)

```
sleep(1.512)
chrono.lap(name = 'step 1')

print('Step two with duration approx. 1.834 secs')
sleep(1.834)
chrono.lap(name = 'step 2')

chrono.compute_durations(ms = True)

print('\nNow printing durations :')
for name, time in chrono.durations.items():
    print('Time for step %s : %d ms' % (name, time))
```

Durations can be written in a tsv file with `Chronos.write_tsv()`. The method uses an append mode, so you can append times from different runs to the same tracking file, for instance. Argument `run_name` in that method allows you to give a name to a run especially for that purpose (the name appears as the first column of the written file).

Three columns are written, with default names `Execution` (the one presented just above), `Step` and `Duration (secs)`. These names can be changed with argument `col_names`.

If you want to time iterations in a for loop, you can use it as a decoration for the iterable. Since you won't be able to assign the object back when the loop finishes, you can choose to print durations in console, or write them into a tsv file.

```
from pyux.time import Chronos
from pyux.time import Timer

for index, value in enumerate(Chronos(range(1, 4), console_print = True, ms = True)):
   Timer(delay = value, message = 'At iteration %d' % index, overwrite = True)
```

Depending on the number of arguments you provide, declaration in the for statement can become rather verbose. Feel free to initiate the chrono outside of the loop, in which case, the object remains available after the loop (if you need to add steps from the rest of the code afterwards, for instance).

```
from pyux.time import Chronos
from time import sleep
from os import unlink

timed_iterable = Chronos(
   iterable = range(25),
   console_print = True,
   write_tsv = True,
   run_name = 'verbose_declaration',
   path = 'example_times.tsv',
   col_names = ('run', 'lap', 'duration (msecs)'),
   ms = True
)
for value in timed_iterable:
    sleep(value / 1000)
# unlink('example_times.tsv')
```

2.2.5 Detailed documentation

class pyux.time.**Chronos**(*iterable=None*, *console_print: bool = False*, *pattern: str = 'iteration '*, *write_tsv: bool = False*, *run_name: str = 'default'*, *path: str = None*, *col_names=None*, *ms: bool = False*)

Bases: `object`

Time chunks of code in a script.

The class works as a stopwatch during a race on track : after starting it, for each *lap*, a button is pushed that records time at this moment, using `timeit.default_timer()`.

Parameters

- **iterable** (*iterable or int*) – default `None` : *for decorator use only* : an iterable, or an integer for standard `range(n)` values.
- **ms** (*bool*) – default `False` : compute durations in milliseconds
- **console_print** (*bool*) – default `False` : *for decorator use only* : print durations in tsv format in console when loop is finished
- **pattern** (*str*) – default `'iteration '` : *for decorator use only* : pattern for naming laps, will be suffixed with the iteration number
- **write_tsv** (*bool*) – default `False` : *for decorator use only* : write out tsv file end loop is finished
- **run_name** (*str*) – default `'default'` : *for decorator use only* : name to give to execution for column Execution
- **path** (*str*) – default `None` : *for decorator use only* : full path to file to write
- **col_names** (*tuple*) – default `None` : *for decorator use only* : column names (length 3)

Recorded times do not have a meaning by themselves since they do not correspond to durations and depends on the previous recorded times. When you want to know duration of recorded laps, use `compute_durations()`, which gives durations for each lap and total duration.

Chronos can be used as a decorator in a for loop. In that case, it records duration for each iteration. Since you cannot get the object back, you can either print the values to console (in tsv format) or write out the values in a tsv file.

Specifying an `iterable` argument while calling `Chronos()` outside of a for statement has no effect. In that case, you can still use it manually, as if you entered nothing for that argument.

Results can be exported in a tsv file with `write_tsv()`.

Example

```
>>> from time import sleep
>>> chrono = Chronos()
>>> sleep(2)
>>> chrono.lap(name = 'lap 1')
>>> sleep(5)
>>> chrono.lap(name = 'lap 2')
>>> chrono.compute_durations().durations
>>> # In a for loop
>>> for index in Chronos(10, console_print = True, write_tsv = False):
>>>     pass
```

compute_durations(*ms: bool = False*)

Compute laps durations.

Duration is the difference between two adjacent laps. Results are stored in `self.durations`, in seconds by default. They can be stored in milliseconds. The total duration is also calculated.

Parameters **ms** (*bool*) – express durations in milliseconds rather than seconds

Returns self

lap(*name: str*)
Record time for this lap; a name must be provided.

write_tsv(*run_name: str*, *path: str*, *col_names: tuple = None*)
Export durations in a tsv file.

Write three columns, the first containing `run_name` : a string describing which execution the durations came from. This way you can append several execution times to the same file.

Default values for column names are : Execution, Step, Duration (secs)

Parameters

- **run_name** (*str*) – name to give to execution for column Execution
- **path** (*str*) – full path to file to write
- **col_names** (*tuple*) – default `None` : column names of length 3

Returns self

class pyux.time.**Timer**(*iterable=None*, *delay: int = None*, *message: str = ''*, *ms: bool = False*, *pattern: str = ''*, *print_value: bool = False*, *overwrite: bool = False*)
Bases: object

Print a descending timer for a given delay.

A message can be printed next to the timer. The class can be used on the iterable of a for loop to wait the same amount of time at each iteration.

Parameters

- **iterable** (*iterable or int*) – default `None` : *for decorator use only* : an iterable, or an integer for standard `range(n)` values
- **delay** (*int*) – default `None` : time to wait, must be provided. Default to seconds, can be milliseconds with `ms = True`
- **message** (*str*) – default `''` : message to print when used manually
- **ms** (*bool*) – default `False` : use a delay in milliseconds
- **pattern** (*str*) – default `''` : *for decorator use only* : prefix to print before iterated value when used as a loop decorator and `print_value = False`
- **print_value** (*bool*) – default `False` : *for decorator use only* : Print running value of the iterable when used to decorate a loop instead of iteration index
- **overwrite** (*bool*) – default `False` : do not print a new line when the timer is finished (useful almost only when used within a loop)

When used within a for loop, the default behavior is to print the iteration index next to the timer. You can add a constant string prefix to it using the argument `pattern` (which is not used otherwise), or use `print_value = True` to print the running value of the iterable.

Specifying a value for `iterable` while calling `Timer()` outside of a for statement will have no effect (except if you use the result in a for statement afterwards, obviously).

Raises `DelayTypeError` if no or wrong type delay is given

Example

```
>>> # One-shot usage
>>> Timer(delay = 10, message = "Waiting for 10 seconds")
>>> # Usage as a decorator
>>> for index in Timer(3, delay = 3):
>>>     pass
```

close (*message: str*, *overwrite: bool*) → None
Print `'0'` when time has passed (last iteration is `1`).

static print (*time: str*, *message: str*) → None
Print the counter for a given time, counter and message.

pyux.time.**timeout** (*delay: float*)
Stop a function if running time exceeds delay (seconds).

To be used as a decorator.

Parameters **delay** (*float*) – time in seconds before stopping running function

Example

```
>>> @timeout(delay = 10)
>>> def infinite_fun():
>>>     while True:
>>>         pass
```

pyux.time.**wait** (*delay: float*, *before: bool = True*, *timer: bool = False*, ***timer_args*)
Wait a given delay before or after a function call.

The function is used as a target function's decoration. With default values nothing is printed during pause, but a `Timer` can be printed instead.

Parameters

- **delay** (*float*) – time to wait (in seconds)
- **before** (*bool*) – default `True` : pause before function execution
- **timer** (*bool*) – default `False` : print a timer during pause
- **timer_args** (see `Timer`) – keyword arguments for `Timer` when `timer = True`

Returns a function decorated with a timer

Example

```
>>> @wait(delay = 3)
>>> def print_hello():
>>>     print('hello')
>>> print_hello()
```

2.3 Logging

2.3.1 Instantiate a logger

The module contains a function `init_logger` that returns a logger from the logging package with a fixed formatting, but choice in the log file name. The default name contains the date and time of execution.

A different log file is created in the given folder at each code run, if the default name for the log file is used. If you set an equal name from one run to another, the various logs will be appended to the same log file.

`pyux` comes with a default format for the logger, but you can specify your own `logging.conf`. Feel free to use `ColourPen` to color logger messages :

```
from pyux.logging import init_logger
from pyux.console import ColourPen
from shutil import rmtree

logger = init_logger(folder = 'logs', filename = 'activity', run_name = 'exemple',␣
↪time_format = '%Y%m%d')
pen = ColourPen

# writes in green for debug
pen.write(color = 'green')
logger.debug('This ia a debug')

# writes in red for critical
pen.write(color = 'red', style = 'bright')
logger.critical('This is a critical')

# go back to normal for info
pen.write(style = 'RESET_ALL')
logger.info('This is an info')

# rmtree('logs')
```

The same logger can be used throughout a whole project by calling `logger = logging.getLogger(__name__)` in submodules of the main script.

2.3.2 Detailed documentation

`pyux.logging.`**`init_logger`**(*folder: str = './logs'*, *filename: str = 'activity'*, *run_name: str = 'default'*, *time_format: str = '%Y%m%d-%Hh%Mm%Ss'*, *config_file=None*)

Return a logger instance with predefined or custom format.

A different log file is saved at each execution. The logger is formatted with a default `logging.conf` when `config_file` is not provided.

Log file name is of the form `filename_run_name_time-format`. Default values yield, for instance : `logs/activity_default_20190721-18h34h20s.log`. You can cheat with `time_format` by specifying a normal word.

To use the logger across submodules, it is advised to instantiate the logger in the main script with `logger = init_logger()`, then instantiate it in the other modules with `logger = logging.getLogger(__name__)`. This will not duplicate logging instances, and will display in the logged message the name of the module from which the logger was called.

Parameters

- **`folder`** (*str*) – default `'./logs'` : folder to save log files in, created if does not exist
- **`filename`** (*str*) – default `'activity'` : name of log files
- **`run_name`** – default `'default'` : name of run
- **`time_format`** (*str*) – default `'%Y%m%d-%Hh%Mm%Ss'` : time format for date
- **`config_file`** (*str*) – default `None` : path to logging.conf file

Type run_name: str

Returns A 'root' logger from `logging` package

Example

```
>>> logger = init_logger(
>>>     folder = 'logs', filename = 'exemple',
>>>     run_name = 'daily-run', time_format = "%Y%m%d"
>>> )
```

2.4 Errors

exception pyux.errors.**ColorValueError**
Bases: `ValueError`

Color is not available for ColourPen.

exception pyux.errors.**DelayTypeError**
Bases: `TypeError`

No delay was given to Timer.

exception pyux.errors.**NoDurationsError**
Bases: `ValueError`

Durations were not yet computed in Chronos.

exception pyux.errors.**StyleValueError**
Bases: `ValueError`

Style is not available for ColourPen.

exception pyux.errors.**TimeoutThreadError**
Bases: `TimeoutError`

Fail to start thread for timeout decorator

2.5 Release notes

2.5.1 Version 0.1.4

- all classes and functions can be imported from root `from pyux import <target>`
- add codacy checks and flake8 enforcements

2.5.2 Version 0.1.3

- `timeout` decorates a function to make it stop after a given time of execution
- modules documentation were moved to their page instead of being in the `README`

2.5.3 Version 0.1.2

- Both `Timer` and `Wheel` now have their decorating equivalent : `wait` and `wordy`
- `wait` decorates a function to make it pause before or after execution
- `wordy` decorates a function to make it say a message after execution, with the ability of catching exceptions
- `Speak` is the iterable version of `wordy` : applied to an iterable in a for loop, it prints a fixed message (or preferably a single character) at each iteration
- a `contributing` section was added in the documentation and the whole documentation and docstrings have been cleaned from some typos

2.5.4 Version 0.1.1

- `Timer` and `Chronos`, even if it is less expected from them, can be used as for loop's decorations too
- Documentation was improved : both docstrings, readthedocs structure and README
- The demonstration was updated to be more efficient in presenting the available tools
- Pipelines set up on GitLab so that `develop` version is always available on readthedocs

2.5.5 Version 0.1.0

- First version published on PyPi
- `ColourPen` has a standalone `write` method, only this one can be used to colourise different sentences
- `Wheel` and `ProgressBar` can be used as for loop's decorations, avoiding to manually declare instantiation, printing and closing

2.6 Contributing

All contributions to the package are welcome ! It is likely that the repository settings on Gitlab are not perfectly tuned for anyone to contribute, so feel free to contact the package's maintainer if you encounter any problem.

Since `pyux` is a simple and little package, there are no real rules when contributing, rather some simple guidelines to keep it clean and sound.

2.6.1 Coding convention

Code

For python's code, we generally follow PEP8 convention, for many of our contributors use PyCharm as IDE, which automatically checks for that convention. As can be seen below, we'd be happy that you add type indications to a function's signature.

Docstrings

We mostly follow the rules given by `sphinx` documentation for docstrings, about which you can read here.

- present tense and imperative mode is preferred for the first sentence, which should end with a period
- always use triple quotes for docstrings (`"""Your line."""`) it makes it

easier for next contributors to add new lines to the docstring. - for multi-line docstrings, please do not leave a starting blank line - specify argument types and return with `:param x:`, `:type x:`,

`:return: something` and `:rtype: str` (for instance)

- try as much as possible to provide at least one usable example
- do not start an argument specification with a cap
- if you write more than one description paragraph, please put all but the first *after* the function's arguments specification, but *before examples*

A typical docstring would thus be :

```
def a_wonderful_function(first_arg: int, second_arg: str) -> None:
    """Print first the second, secondly the first."""
    print("A string %s with a number %d" % (second_arg, first_arg))
    return None

def another_function(one_arg: str, prefix: str = 'a prefix : ') -> str:
    """Add a prefix to a string.

    This function is not really useful.

    :param one_arg: the string to be prefixed
    :type one_arg: str
    :param prefix: default ``'a prefix'`` : the prefix to use
    :type prefix: str
    :return: the input string prefixed with the given prefix
    :rtype: str

    Here goes the rest of the description, details, etc.

    :Example:

    >>> another_function('rosso', 'pesto')
    """
    return None
```

2.6.2 Pull requests

A default template is provided for merge requests. It is not compulsory to use it, it is only meant to ease reviewing your changes and keeping things simple and straightforward.

2.6.3 Commit conventions

The more your commits will follow these conventions, the easier it will be to understand what you did and navigate through the repo's history.

- use present tense and imperative mode for the first line of your commit : instead of “Added new function” or “Adds new function”, write “Add new function”
- keep the first line less than 80 characters and without period at the end
- feel free to explain with as many lines as needed after the first line, but always leave the second one empty
- using emojis to prefix commits is highly appreciated, not only for fun but also because it adds another layer of easily understood contextualisation

About emojis, feel free to use whichever you want to, here are the ones that are the most often used in the package :

- or When your changes improve performance
- When you modify comments, docstrings or non-python files
- When you modify the documentation files (in `docs/source`)
- When you change or fix some object mechanism
- When you refactor something (more than a little change)
- When you resolve a bug starting from `develop`
- When you resolve a bug starting from `master`
- When you delete or remove something (in the code or a whole file)
- When you improve format and code’s appearence (indentation, etc.)
- or When you move files upward or downward in the folder’s tree
- and When you move files from a folder to another in the same level of the folder’s tree
- (or any clock) When the changes are temporary or need to be refined
- When the changes are a work in progress and call for subsequent changes
- When you add tests or make broken tests work
- or When some work or improvement has been done (in short, when none of the above apply)
- When a new functionality has been completed
- and Reserved for releases of merge commits
- when you did something you’re very proud of

CHAPTER 3

genindex

Python Module Index

p

Index